Contraction Hierarchies Algorithm: Implementation and Experiments

CS780 Project Report

Deqi Li

Department of Computer Science, University of Auckland, Auckland, New Zealand

(February 17, 2014)

Abstract. Contract Hierarchies algorithm (abbreviated for CH algorithm), presented by Geisberger et al. in 2008 [1, 2, 6], is one of currently most efficient algorithms for finding the shortest path on a graph. Although its core is still Dijkstra's algorithm, it dramatically improves the later's performance by adding shortcuts and pruning searching space. CH algorithm has two phases: graph preprocessing and shortest path query. On the preprocessing phase it adds shortcuts into the graph and generates orders of nodes. On the query phase, it searches the shortest path using bi-directional Dijkstra's algorithm and in each direction it explores nodes strictly in ascending order. The preprocessing phase is heuristic, but query phase is deterministic and always gives optimal results. We realized CH algorithm and tested its performance by experiments as well as proved the core ideas applied to the CH algorithm. By comparing CH algorithm with the single-directional and the bi-directional Dijkstra's algorithm algorithms, we basically confirmed the performance claimed by the inventors of CH algorithm. It is claimed that CH algorithm is one to two orders of magnitude faster than Dijkstra's algorithm. Our goal of this project is to verify the speedup. The CH algorithm is particularly suitable for sparse large planar graphs like road networks. Our experiments are conducted on USA road networks presented by U.S. Census Bureau [12], which comprises of 24 million nodes and 58 million directed edges.

1 Introduction

1.1 Applications and Problems of Dijkstra's algorithm

Finding shortest paths on maps (i.e., map routing) has wide applications such as vehicle navigation and Google maps. In these applications, the maps usually have millions of

nodes and the query is required to be real time. So the time consumption and the space consumption both are very important considerations when we invent or improve query algorithms. Although the famous Dijkstra's algorithm has a good time complexity $O((m+n)\log n)$ or even $O(m+n\log n)$ when we use binary heaps and Fibonacci heaps respectively [9], it still cannot satisfy the requirement of real time applications. Our experiments show that a common modern computer needs more than 7.5 seconds using the bi-directional Dijkstra's algorithm to find a shortest path on the USA road networks (Table 4), which is unacceptable to users. The root of low performance of Dijkstra's algorithm lies in that a random query needs to visit about 10% to 90% nodes of the given graph. Our experiments show that the average number of nodes visited by the bi-directional Dijkstra's algorithm on the road network file"USA-road-d.USA" is 8.6 million, which is about 36% of the whole nodes. Visiting such a large number of nodes is very time consuming.

The following are some terms that we define for reference.

Definition 1. *Starting nodes:* Starting nodes refer to the source node or the target node. If a node is visited by the forward direction, then its starting node is the source node; otherwise if it is visited by the reverse direction, its starting node is the target node.

Definition 2. *Forward direction* and *Reverse direction:* In the bi-directional Dijkstra's algorithm, the algorithm alternates the search from two directions — from the source and from the destination — simultaneously. The former direction is the forward direction and the later is the reverse direction. The later is also named the backward direction in some literatures.

Definition 3. *Tentative distance:* When a node v is visited by the current settling node, its distance to the starting node could be updated to a smaller new value. If tentative distance $>$ d(starting node, settling node) + weight(settling node, v), then tentative distance $=$ d(starting node, settling node) + weight(settling node, v). At initializing phase, except the two starting nodes of which tentative distances are set to 0, the tentative distances from all other nodes to their starting nodes are set to infinite.

Definition 4. *Settling:* A node is settling means it was just on the top of the priority queue and pops out of the queue. It has had a tentative distance to its starting node and now the distance become determined and will not change any more.

Definition 5. *Visited / Reached:* A node is visited (or reached) means it is visited by the current settling node, which is one of its neighbors, and its distance to its starting node is set to a tentative value based on the current settling node.

Definition 6. *Settled / Scanned / Labeled:* A node is settled (or scanned, labeled) means its distance to its starting node is determined after it pops out of the priority queue. Before it is settled, it must have been visited by the search algorithm.

Definition 7. *Search tree:* If we connect each visited node to its previous, we can find that the searching space of Dijkstra's algorithm actually is a tree. Such a tree is called search tree.

1.2 Solutions: Contraction Hierarchies algorithm

Contraction Hierarchies (CH) algorithm is one of currently the most efficient algorithms to solve NSSP (non-negative single-source shortest path problem). It was developed by Geisberger et al. in 2008 [1, 2] to speed up point-to-point shortest path finding for continental-sized road networks. It involves two phases: the preprocessing phase and the query phase. The former is relatively more time-consuming than the later and the efficiency of CH algorithm is gained by amortizing the preprocessing time over many NSSP queries. In the CH preprocessing phase, by the technique node contraction, nearly as the same number as the original graph of shortcuts are added into the graph, and each node is set to a unique order represented its priority during the query phase.

The CH query algorithm, quite similar to the traditional bi-directional Dijkstra's algorithm, is also bi-directional, that is, the query alternates to explore nodes in two directions from the source and from the target respectively. The critical difference is that the bi-directional Dijkstra's algorithm does not prune any nodes, however CH algorithm only search nodes in ascending order in both directions until it satisfies a stopping criterion. Metaphorically, CH query algorithm is such an algorithm that jumps on shortcuts steered by node orders. Thus CH algorithm dramatically reduces the number of visited nodes in the query phase and saves much query time.

1.3 Outline of this report

The following sections of this report are arranged as: Section 2 explains Dijkstra's algorithm and the bi-directional Dijkstra's algorithm as well as its stopping criteria. Section 3 illustrates the details of CH algorithm and its implementation. The experiments to compare CH algorithm and the bi-directional Dijkstra's algorithm as well as the single-directional Dijkstra's algorithm are in Section 4. Section 5 is the conclusion of this report and future work, including a few of the author's observations and new ideas on shortest path searching algorithms.

2 Single-directional and bi-directional Dijkstra's algorithm algorithms

2.1 Single-directional Dijkstra's algorithm

The single-directional DJ algorithm [3, 9] searches the destination from source in only one direction until the algorithm finds the destination. The algorithm works as follows. At first, it selects the source as the first settled nodes, sets its distance to 0 and the distances of all other nodes to infinite. Push this node into a heap. This node is the current minimum node. Pop it out of heap. Based on the minimum node, each of its neighbor v except that have been settled is assigned a new tentative distance if the previous tentative distance is larger than the new one. If its previous tentative distance is infinite, it means that it is the first time that this node is been visited and we push it into the heap; otherwise we just update its tentative distance. When we implement the

heap, it is actually a priority queue that the minimum node is always on the top. It can have various implementations such as binary heaps, Fibonacci heaps [9] and we even use a simple vector. But in order to support Dijkstra's algorithm, these heaps should have at least such five functions: isEmpty() to determine if the heap is empty, removeMin() to get and remove the minimum node from the heap, insert() to insert a new node into the heap, decreaseKey() to decrease the key value (i.e., tentative distance) of a node, and finally, getMin() to get the minimum node of the heap. If we do not consider operations efficiency, a simple vector can support all these operations using brute forcing searching. Interestingly, we will see that for CH algorithm, such a simple vector can surprisingly work very well and even is comparable to binary heaps and more complicated Fibonacci heaps.

Now we are ready to describe the algorithm.

Single-directional Dijkstra's algorithm:
Input: a non-negative weighted graph $G = (V, E)$, $n = |V|$, $m = |E|$; source $s \in V$, target $t \in V$.
Output: the shortest distance between s and t.

```
for i = 1 to n, do //Set initial distances all nodes to infinite
    distance[i] ← ∞
    state[i] ← INITIAL
end for
distance[s] ← 0
Queue.insert(s)
while not Queue.isEmpty() do
    u ← Queue.removeMin()
    state[u] ← SETTLED
    if u = t then
        return distance[u] // the shortest distance
    end if
    for v = each neighbor of u, do
        if state[v] = SETTLED then
            next v
        end if
        tentative ← distance[u] + weight(u, v)
        if tentative < distance[v] then
            previous[v] ← u
            if distance[v] = ∞ then
                distance[v] ← tentative
                Queue.insert(v)
            else
                distance[v] ← tentative
                Queue.decreaseKey(v, tentative)
            end if
        end if
        next v
    end for
end while
return distance[u] // the shortest distance
```

The algorithm above only gives the optimal distance between s and t but it does not show the path. In fact, the path information is already available in the previous array for each node v when v updates its tentative distance. If we trace back previous nodes starting from the destination until the source, we can easily composite this path (see CH showPath algorithm in Section 3.4.4).

2.2 bi-directional Dijkstra's algorithm

2.2.1 How it works

The bi-directional Dijkstra's algorithm [4] searches from source and destination simultaneously. The searches are called the forward search and the reverse search (or the reverse search) respectively. At the first step, the distances of source and destination are set to 0 and push them into heaps. During the search, some nodes can be visited by the two directions, thus we need two heaps to store the node information such as distances and states. So it is quite like two paralleling single-directional Dijkstra's algorithm. And in the implementation, we alternate the forward search and the reverse search. The following is the bi-directional Dijkstra's algorithm, where $Queue_1$ and $Queue_2$ are two priority queues, Q_f and Q_r are the settled distances of the minimum nodes (sometimes they also denote the nodes on the top of the priority queues), *best* is the optimal distance from the source to the target seen so far, and $flag$ indicates the search direction.

2.2.2 Why two-direction searches reduce the searching space

When Dijkstra's algorithm is searching the destination, it does not know the accurate direction of the destination, so it has to explore nodes in all directions. Thus the searching space is roughly a circle with the source as the center. The number of nodes it visits is proportional to the square of the radius of the circle, i.e., the area of the searching space. If we search in two directions, roughly the two searching trees meet at the middle of the graph (note: this is not always the case). Intuitively, the radii of two searching spaces are only half size of that of the single-directional Dijkstra's algorithm. Thus a simple calculation tells us that the two searching space together are only half of that of the single directional Dijkstra's algorithm, that is, we reduce the searching space by half.

2.2.3 the Speedup

Our experiments show that usually the bi-directional Dijkstra's algorithm is about 1.2 – 2 times faster than the single-directional version (see Table 4 in Section 4: Experiments).

2.2.4 Update the best distance

Let Q_f be the distance from the top node in the forward priority queue to the source s, Q_r the distance from the top node in the reverse priority queue to the target t, μ the optimal distance between s and t seen so far (precisely, it is the optimal distance after the previous Q_f or Q_r updates their neighbors), $d(s, v)$ the distance from node v to s, $d(w, t)$ the distance from node w to t, and weight (v, w) the weight of the edge (v, w). Sometimes we also use Q_f and Q_r to represent the nodes on the tops of the two priority queues. Whenever a node u is settling (it means that it is just popped up from a priority queue) in the forward direction, it always tries to update the tentative distances of its neighbors v in the forward direction. Meanwhile, if one of v's neighbors w has been settled by the reverse direction (this means v has been visited by w, so v has had tentative distance to

the target in the reverse direction. See Figure 1), and $d(s,v) + \text{weight}(v,w) + d(w,t) < \mu$, then μ is updated at node v:

$$\mu = d(s,v) + \text{weight}(v,w) + d(w,t)$$

Figure1 shows μ is being updated at node v when node u is settling and visiting v.

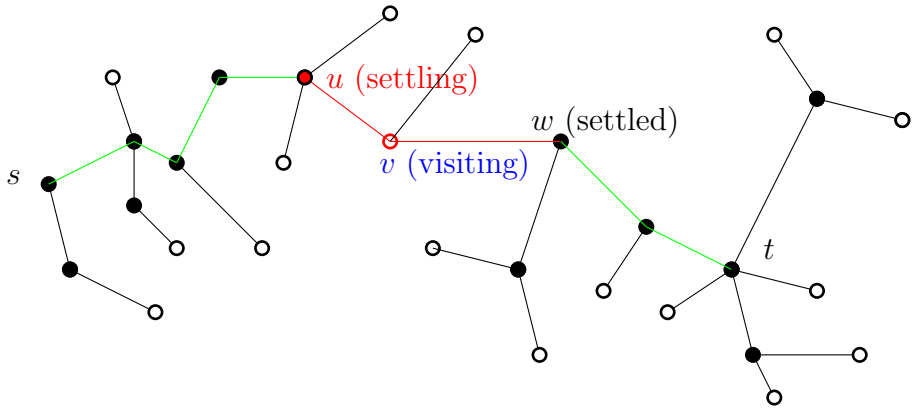

Figure 1: Update μ when two search trees meet at v.

Similarly, we also update μ in the reverse direction.

2.2.5 Stopping criteria (termination condition)

How to determine when or where we should stop the algorithm is the major problem of the bi-directional Dijkstra's algorithm. In history various stopping criteria were presented but proved wrong [16], so the correctness of stopping criteria is not obvious. This paper introduces two stopping criteria: the weak stopping criterion and the strong stopping criterion [16, 17].

1. the weak stopping criterion

   Intuitively we can immediately stop the algorithm once the two search trees meet somewhere.

   Definition 8. *the weak stopping criterion (version 0):* The bi-directional Dijkstra's algorithm can stop immediately once two search trees meet at some node.

   Theorem 9. *The weak stopping criterion is not always correct.*

   We found that the weak stopping criterion is not strictly correct even if we understand the word meet" in any case in the following. Note that we should not use the vague term "meet"; instead, we use the terms such as "reach" (or "visit" in some literatures) and "settle" (or "scan", "label") that we have defined. We will explain this theorem in detail.

   For example, we understand it as "some node is visited twice".

Definition 10. *the weak stopping criterion (version 0, specified):* The bi-directional Dijkstra's algorithm can stop immediately once some node is visited twice (Figure 2).

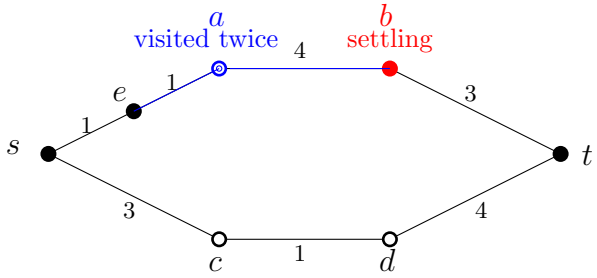

Figure 2: node a is visited twice

Here is an example to illustrate this misunderstanding. As shown in Figure 2, when node a is being visited by the reverse direction, it is visited twice because it has been visited by the forward direction. But we cannot stop now; otherwise the shortest path would be s-a-b-t with length 9, however the real shortest path is s-c-d-t with length 8.

The second understanding of "meet":

Definition 11. *the weak stopping criterion (version 1):* The bi-directional Dijkstra's algorithm can stop immediately once some node that has been settled in one direction is reached in another direction [16].

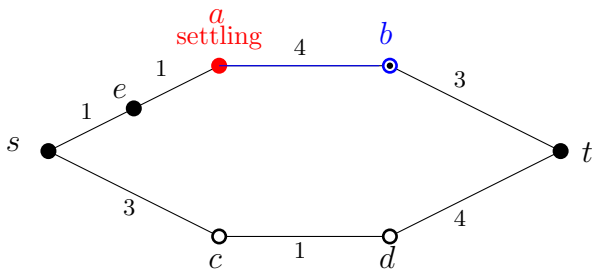

Figure 3: node b has been settled in the reverse direction and is being visited in the forward direction

As shown in Figure 3, the node b has been settled by the forward search and it is also visited by the reverse search. But we cannot stop now; otherwise the shortest path would be s-a-b-t with length 9, however the real shortest path is s-c-d-t with length 8.

The third understanding of "meet":

Definition 12. *the weak stopping criterion (version 2):* The bi-directional Dijkstra's algorithm can stop immediately once some node is settled twice.

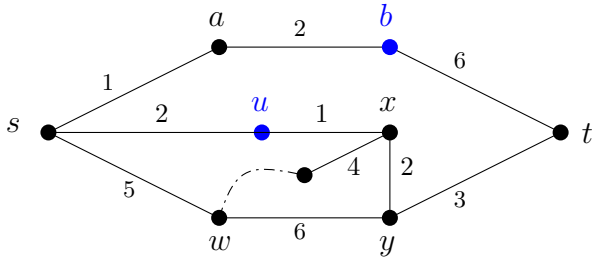

Figure 4: Counterexample 1: node b or u is going to be settled twice.

In Figure 4, in the forward direction, s, a, b, u are settled; x, w are visited and they are in the forward priority queue. In the reserve direction, t, y, x are settled, and either b or u is settled. The following is a counterexample of the weak stopping criterion (version 2). The two directions settled nodes in the following sequences (the number in brackets are distances of nodes to their starting nodes):

In the forward direction: $s(0), a(1), u(2), b(3)$;
In the reverse direction: $t(0), y(3), x(5), b(6)$ or $u(6)$.

After the reverse direction settled x, it has two options: to settle b or to settle u, because these two nodes have the same distances to t, so each one of them are possibly at the top of the reverse priority queue. Each of them also can trigger the weak stopping. If the queue pops u, then the shortest distance from s to t is 8; however, if the queue pops b, then the shortest distance is 9. Obviously 9 is a wrong result.

The Figure 5 shows another counterexample for the weak stopping criterion (version 2).

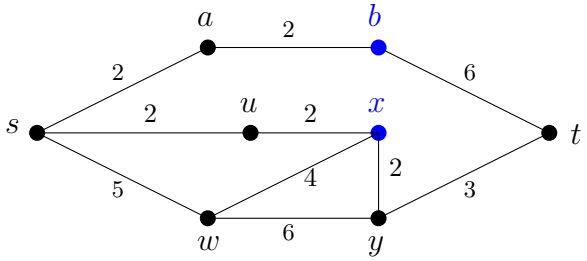

Figure 5: Counterexample 2: node b or x is going to be settled twice.

The two directions settled nodes in the following sequences:

In the forward direction: $s(0), a(2)$ or $u(2), u(2)$ or $a(2), b(4)$ or $x(4)$;
In the reverse direction: $t(0), y(3), x(5), b(6)$.

Node b and node x both are possibly settled twice, but if b is settled twice first, then the result is 10, which is wrong (the shortest distance is 9).

To sum up, the three definitions above of the weak stopping criterion are not always correct (therefore, it is NOT a special case of the strong stopping criterion as following).

2. the strong stopping criterion

If $Q_f + Q_r >= \mu$, then the algorithm can stop and return μ as the shortest distance between s and t [16, 17].

Recall that μ is the best distance from source to destination, Q_f and Q_r are the distance of the top node in the priority queues, s and t are source node and target node respectively. Unlike the weak stopping criterion, the strong stopping is correct. Before we prove this criterion, firstly we make some observations and prove two theorems. We made the following three observations which we use them as lemmas.

Lemma 13. *Q_f and Q_r are always non-decreasing; that is, the distance of the minimum node on the top of the priority queue to its starting node in each direction is always increasing.*

Lemma 14. *In the forward direction, the nodes that have been popped and settled have distances less than Q_f; similarly, this holds in the reverse direction.*

Lemma 15. *In the forward direction, the nodes that have NOT been popped and are still in the forward priority queue have distances larger than Q_f; similarly, this holds in the reverse direction.*

The three lemmas above are keys to understand the strong stopping criteria. Our proof of the strong stopping is as follows.

Proof. Suppose at the moment τ, the condition $Q_f + Q_r >= \mu$ is satisfied. It is enough that we can prove that μ will not be updated after the moment τ when any new path is formed (that is, we will ignore the paths that were found before τ). But for convenience, we inspect a process starting from a little earlier moment $\tau_0 < \tau$, and at τ_0, μ is updated for the last time before τ.

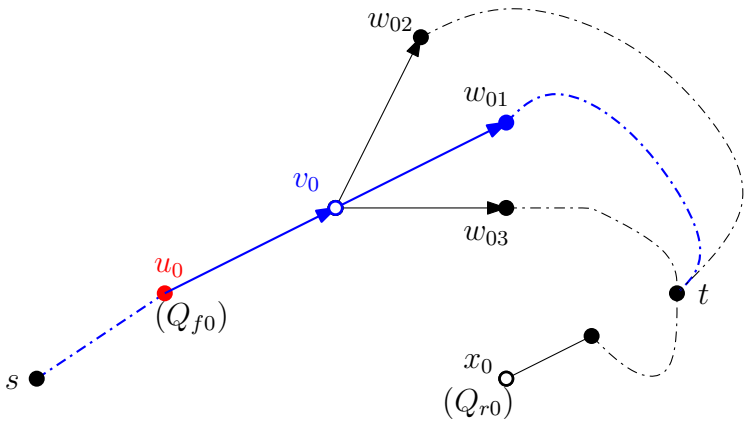

Figure 6: μ is updated from μ_0' to μ_0 at node v_0 for the last time at the moment τ_0 (before τ). The blue path is the new path of length $\mu_0 < \mu_0'$.

At the moment τ_0 (Figure 6): The forward direction and the reverse direction meet at v_0; A new path going through the edge(u_0, v_0) is found (actually, maybe more than one paths are found, but they all go through the edge(u_0, v_0)). μ is updated

from μ_0' to μ_0:
μ: $\mu_0' \to \mu_0$.

$(Q_{f0} + Q_{r0} < \mu_0')$

During the period (τ_0, τ): The two search trees do not meet each other. μ is not updated. So $\mu = \mu_0$.

At the moment τ (Figure 7): $Q_f + Q_r \geq \mu_0$.

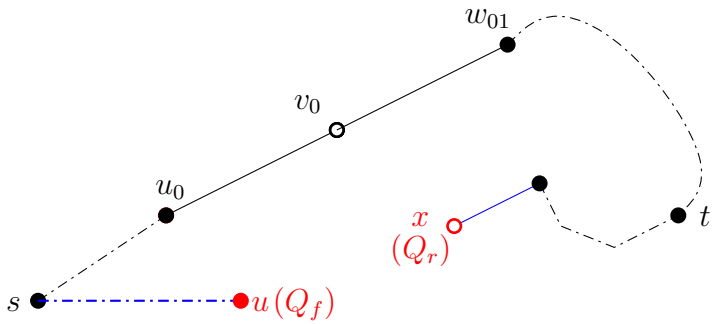

Figure 7: $Q_f + Q_r$ surpasses μ_0.

At the moment $\tau' > \tau$: A new path going through (u', v') is formed(actually, maybe more than one path are found at this moment). Let $d(s, u)$ denote the shortest distance from s to u', weight (u', v') the weight of edge(u', v'), and tentative(v', t) the tentative distance (not finally determined yet, so could be updated later on) from v' to t. See the CH query algorithm (in Section 3) and Figure 8.

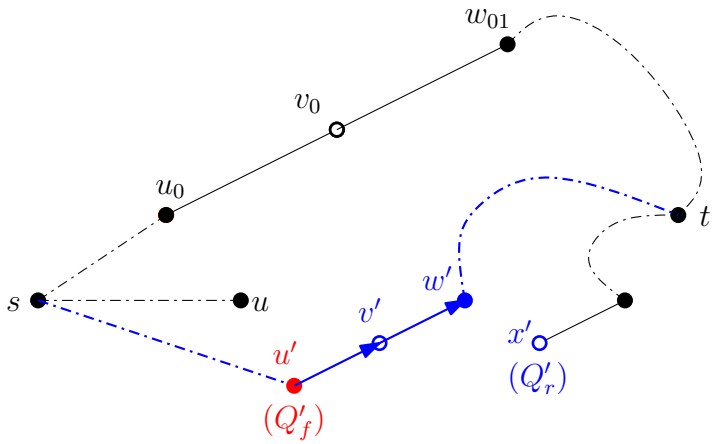

Figure 8: The two search trees meet again after the moment τ, but μ will not be updated.

There are two cases corresponding to settle a node in the forward direction or in the reverse direction.

Case 1: the algorithm is settling node u' in the forward direction. Now u' is visiting one of its neighbor v'.

If v' has not been visited in the reverse direction, then its tentative $(v', t) = \infty$, and no path is formed at the moment τ'.

If v' has been visited but not settled yet in the reverse direction, then a new path $< s, \ldots, u', v', \ldots, t >$ going through the edge(u', v') is found at τ'. We notate the length of this candidate path ℓ'. We have $\ell' = d(s, u') + \text{weight}(u', v') + \text{tentative}(v', t)$ and $Q'_f = d(s, u')$. According to Lemma 13, $Q'_f \geq Q_f$. According to Lemma 15, tentative $(v', t) \geq Q'_r \geq Qr$, $\text{weight}(u', v') > 0$. So, $\ell' \geq Q'_f + \text{weight}(u', v') + Q'_r \geq Q_f + \text{weight}(u', v') + Q_r > Q_f + Q_r > \mu_0$. Thus μ will not update; it is still μ_0.

If v' has been settled in the reverse direction, then the path $< s, \ldots, u', v', \ldots, t >$ has been found before the moment τ' when v' was visited for the second time in either direction.

We will see that the length of this path does not change once it was found (before v' has been settled), because when the two search trees meet at v', v' was visited by, suppose w', in the reverse direction (actually maybe it was visited by more than one nodes in the reverse direction and thus it was associated to multiple paths; now we can pick one of them for inspection). Note that w' was settling and its distance to t was determined rather than tentative. So the length of the path going through u', v', w' satisfies that:
$\ell' = d(s, u') + \text{weight}(u', v') + \text{tentative}(v', t') = d(s, u') + \text{weight}(u', v') + \text{weight}(u', v') + d(w', t)$.
The four items on the right side are all determined, so ℓ' does not change after the path was formed.

Case 2: the algorithm is settling in the reverse direction.
Similarly, we can prove the criterion when some node is settling in this direction.

To sum up, μ will not be updated after the moment τ. Therefore we can stop the algorithm at the moment τ when $Q_f + Q_r \geq \mu$.

□

3 Contraction Hierarchies Algorithm

3.1 CH preprocessing algorithm

Given a non-negative weighted directed graph $G = (V, E)$, where V is the node set of G, E the edge set of G, and a source node s, a target node t, our goal is to find a shortest (optimal) path and its distance from s to t (sometimes the path is not unique). In this report, we not only consider the computation of the shortest distance but also the composition of the specific path. The traditional Dijkstra's algorithm searches many nodes when finding the shortest path, in contrast, Contraction Hierarchies (CH) algorithm only explore a very small fraction of nodes because it searches only in the node order ascending direction in both the forward direction and the reverse direction. The node order is named "query order", which is one of the most important terms in CH algorithm. The goal of CH preprocessing phase is to output an appropriate query order such that CH query can be highly efficient.

At the preprocessing phase, CH algorithm initially computes the "importance" of each

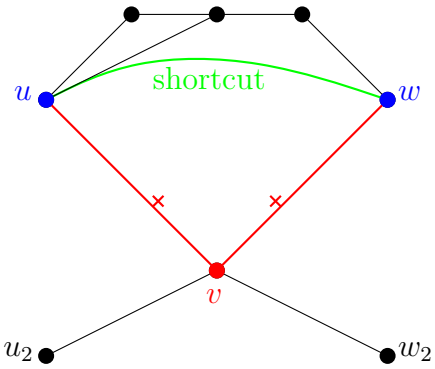

Figure 9: Node contraction: contract node v, remove edges(u, v) , (v, w) and add shortcut (u, w) if necessary. Node order of v is smaller than that of u and w.

node. Then in the order of the importance CH algorithm contracts each node v (actually the algorithm does not really remove nodes from the graph; it only labels their states CONTRACTED; in the query phase the state is ignored). Suppose u and w are neighbors of v. To preserve the shortest path between any node pair of the graph, for a pair $< u, w >$, if $d(u, w) >$ weight(u, v) + weight(v, w) after v is contracted (that means the shortest path $< u, \ldots, w >$ does not go through v), then CH adds a shortcut (u, w) with the weight weight(u, v) + weight(v, w) if the edge (u, w) does not exist in the graph, or only updates weight of (u, w) if (u, w) is an existing edge of graph. The idea of node contraction is shown in Figure 9.

Theorem 16. *CH preprocessing algorithm does not change the (shortest) distance between any pair of nodes of the graph even if we do not consider contracted nodes when we compute the distance.*

Proof. During the CH preprocessing, what we do to contract nodes includes:

1. Set the selection ordering for node contraction and the query order for each node;
2. Update the weight of each pair of nodes (u, w) if a path from u to w is shorter than weight(u, w);
3. Add a shortcut (u, w) if weight(u, v) + weight(v, w) is less than the shortest distance of all paths not going through v when we contract node v.

Obviously, Step 1 does nothing with any distance because it does not really contract nodes and update weights of edges. Step 2 just uses the existing shorter path from u to w (actually it is the shortest path if we adopt unlimited-hop local search to determine the distance) to update a longer path from u to w, but it does not create a new shorter path from u to w in the graph, so the shortest distance $d(u, w)$ does not change. Step 3 essentially is the same case as Step 2, i.e., from u to w it just creates a new path with its length being the same as the previously existing shortest path.

Ignoring contracted nodes does not change the conclusion above because when a node is contracted, we always preserve the distances between each pair of its neighbors by adding new shortcuts or update weights of each pair. This is the key to understand why CH query algorithm can prune nodes of which the query order is lower than the current

settling node. To sum up, we do not change the distance between any pair of nodes. This is the foundation of node contraction.

□

Note that we have to determine whether we should add a shortcut for each pair of the neighbors of v (this process is called "finding witness path"), thus we need to compute many distances like $d(u, w)$ when the degree of v is large. Actually, as more and more shortcuts are added into the graph, the average degree is increasing. At the later phase of CH preprocessing, the degrees of some nodes could be over 50, which are 20 times of the original degrees. Therefore, finding witness paths is a very time-consuming process. Theoretical analysis shows that an optimal design of CH preprocessing that resulting the minimal query time and the smallest number of shortcuts is an NP-hard problem [2]. In practice, we use a staged restarting approach and uniform node selection to initialize the importance of node to avoid the case that the graph becomes too dense. This approach is heuristic. We also presents some novel methods to strike with the time consumption. Our methods include the exponentially decreasing stage array for staged restarting and the single-source-multiple-destination Dijkstra's algorithm to find witness paths for multiple node pairs simultaneously. The following subsection is our implementation details of CH preprocessing algorithm.

3.2 Implementation of CH preprocessing algorithm

3.2.1 Priority terms and node selecting order

Which node is selected first and which one is selected second to contract is important because it leads to different preprocessing time and even impacts the query time. This is the issue of selecting order of nodes. The selecting order is represented as the sum of some priority terms [1, 2]. There are various priority terms and we can use one of their combinations, and the most common and important two priority terms are edge difference E and contracted neighbors D [1, 2]. E is defined as the number difference of the edges of the graph before and after contracting a node, specifically, E is the number of shortcuts to be added due to contraction of a node subtracting the number of the edges incident to this node. If we also use D as one priority term, then as we contracting nodes, the order of the remaining nodes are increasing because their neighbors are being contracted. In this report, we use the two priority terms as the node selecting order during the node contracting process. They have different coefficients. We define selecting order as follows [1, 2],

$$\text{selecting order} = 190 \times E - 120 \times D$$

Note:

1. This selecting order is a different term to the query order which is the output of the preprocessing phase.
2. More complicated combinations of priority terms could result in less preprocessing time and query time.

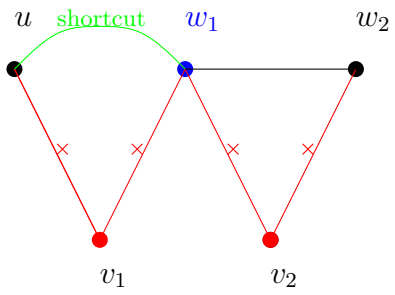

Figure 10: why we need restart: selecting order is always changing during node contraction process. The blue node w_1 loses two original edges and add a new shortcut edge.

All the papers available about CH algorithm ([1, 2, 5, 7, 10, 11, 13]) only present high-level concept of contraction hierarchies algorithm but they do not give the implementation details. Therefore we have to design the details of our implementation. We designed an exponentially decreasing partition of nodes (see the array $Stage[37]$ in Appendix: Code). $Stage[37]$ represent 37 restarting points for 37 contracting stages with the property that

$$Stage[i+8] - Stage[i+4] = \tfrac{1}{2}\ (Stage[i+4] - Stage[i])$$

So roughly, the number of nodes contracted in each stage exponentially decreases. The purpose of such design is to trade off the exponentially increasing shortcuts and the resulting exponentially increasing computing time for finding witness path. After restarting, we can contract nodes starting from nodes of low selecting orders.

Once we have contracted $Stage[i]$ nodes, we restart an initial procedure to recompute the selecting order of nodes that have not been contracted, and re-sort all of them in ascending selecting order, then begin node contraction of the next stage.

The reason of restarting is that, although we have sorted them at the beginning of each stage, during the contraction process, the selecting orders of the remaining nodes are always changing because, as we mentioned, the priorities (i.e., selecting orders) of the neighbors of a contracted node are updated and they could have more neighbors as new shortcuts are added as shown in Figure 10. The restarting allows us to select nodes that have lower priorities (this means it need add less shortcuts and it have more edges) which consume less time than those with higher priorities; and meanwhile it allows us to defer the contraction of these nodes with higher priorities to the later preprocessing phase; because the number of their neighbors could be reduced as their neighbors are contracted (notice that contracted neighbors are not be considered when finding witness paths, because, firstly, as mentioned before, ignoring contracted nodes does not change the distance of any pair of their neighbors; secondly, we can reduce computation for finding witness paths when the node to be contracted has fewer neighbors to be considered). In a word, the purpose of restarting at stages is to keep the graph as sparse as possible so that we can compute for fewer.

This design has overcome the problem that the graph would become too dense at the later phase. On our local server CDMTCS, of which CPU is Intel Core i7-2600 3.4GHz (see Section 4), the preprocessing on the road networks file "USA-road-d.USA" takes 3.3

hours. In contrast, a trivial design makes the program get stuck in preprocessing a small road networks such as "USA-road-d.NY" and consume unacceptably much time.

The following is CH preprocessing algorithm.

CH preprocessing algorithm
Input: road networks graph $G = (V, E), n = |V|, m = |E|$
Output: node orders for query phase

// initial selecting order for contraction
initialStage() // set initial stage array to be n/8, n/8,n/8, n/16+7*n/8,... See Appendix: Code.
$j \leftarrow 0$
$i \leftarrow Stage[j]$
while $i \leq n$
    if $i = Stage[j]$ **then**
        for $k = Stage[j] to n$ **do**
            $SelectingOrder[k] \leftarrow 190 \times E[k] - 120 \times D[k]$
        next k
        $j \leftarrow j + 1$
    end if
    sortSelectingOrder() // sort the selecting order in ascending order
    for $i = stage[j] to stage[j + 1] - 1$ **do**
// select $Stage[i]$ nodes that in ascending order from the remaining nodes
        nodeContract(i)
        updateneighbors(i) // change the selecting priority of neighours
        $order[i] = i$ // the query order of node i for the query phase
    next i
    $j \leftarrow j + 1$
end while
return $order[]$, shortcuts

In Appendix: Code we can see the details of the procedures sortSelectingOrder(), nodeContract() and updateneighbors().

Another quite trivial way to restart at stages is to set the restarting array $Stage[]$ to be linearly increasing numbers rather than exponentially increasing numbers. The author compared the performances of these two approaches and found that the trivial method is so inefficient that it is stuck in the later preprocessing phase because too many shortcuts are added into graph and some nodes have many neighbors.

3.2.2 Witness path finding and limited-hop local search

Even we use the so called staged preprocessing technique with the array $Stage[37]$, the preprocessing is still not efficient enough. Witness path finding in nodeContract() is the most time-consuming in the preprocessing phase. Witness path finding is a local search to find whether there exists a path between any pair (u, w) that does not go through v and is shorter than weight $(u, v)+$ weight(v, w) where v is the node to be contracted and u, w are neighbors of v. If such a path exists, we do not need to add a shortcut (u, w).

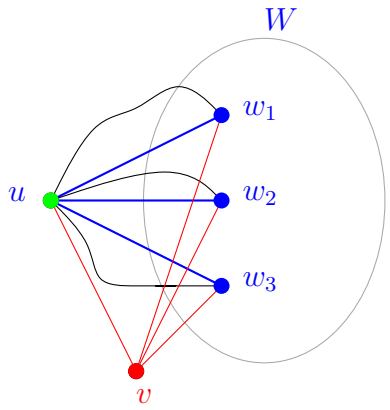

Figure 11: One-source-multiple-destination Dijkstra's algorithm. One source: u, multiple destinations: W.

The paper [1] introduce limited-hop local Dijkstra's algorithm to find witness paths. In this paper, the number of hops initially is 1. When the graph becomes dense, the preprocessing algorithm increases the number of hops from 1 up to 6 at the later preprocessing phase. Note that sometimes this approach cannot find an existing witness path due to its limited hops. But if we use unlimited-hop local search, it could consume much time to find a witness path. However, we propose an unlimited-hop witness path finding approach using single-source-multiple-destination single-directional Dijkstra's algorithm, which dramatically reduces search time by avoiding repeated computation.

3.2.3 Unlimited-hop single-source-multiple-destination Dijkstra's algorithm

We develop this new technique to improve the limited-hop local search. Suppose we are contracting node v. The " single-source" u is a neighbor node of v, and the "multiple-destination" is all other neighbors of v except u (we notate this set W). It returns multiple distances responding to the "multiple-destination" rather than only one destination, but it does not use much more time compared to one run of "single-source-single-destination" algorithm.

The efficiency of this technique lies in that this design can avoid much repeated computation. Notice that $w_1, w_2, \ldots, w_i \in W$ are local nodes close to each other (only 2-hop reachable to each other via v, even they are directly connected), once we have computed the distance $d(u, w_1)$ using Dijkstra's algorithm, we probably also have searched the search space for finding the distance $d(u, w_2), d(u, w_3), \ldots$, because the search space of Dijkstra's algorithm is a search tree and two local search trees usually have much overlap as shown in Figure 12. From this figure, separated runnings of Dijkstra's algorithm local searches generate much overlapped search space. However, single-source-multiple-destination Dijkstra's algorithm can avoid this issue: starting from u, it searches the shortest paths for all destinations $w_1, w_2, \ldots, w_i \in W$, until all destinations are found. Note that the overlapped space is computed only once.

The single-source-multiple-destination Dijkstra's algorithm are nearly d times faster than its single-destination version, where d is the degree of u. So this technique becomes even more important when the graph becomes denser when more shortcuts are added.

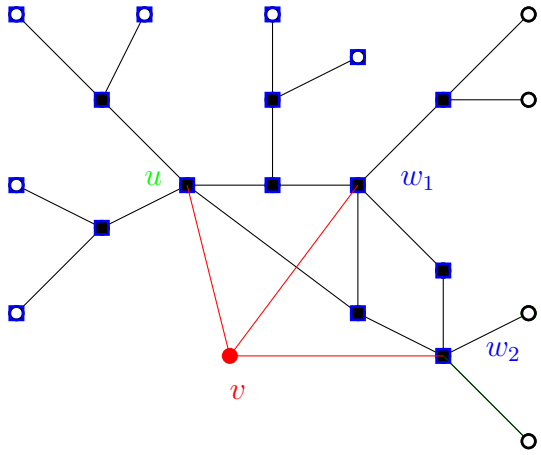

Figure 12: Overlapped search trees: blue nodes are explored twice when Dijkstra's algorithm searches $d(u, w_1)$ and $d(u, w_2)$.

Another advantage of this technique is that the distances it finds is optimal because it does not limit the number of hops. In contrast, the local search method of limited hops used by the paper [1] sometimes could not find the shortest distance.

3.3 CH query algorithm

3.3.1 Two modifications

If readers are familiar with the bi-diretional Dijkstra's algorithm, CH query algorithm is quite simple and straightforward. It is just based on the traditional bi-directional Dijkstra's algorithm and makes two major modifications. One is that it prunes the search space with node orders. In both the forward direction and the reverse direction, only nodes that with orders larger than the current settling nodes are visited. Another modification is that it uses a different stopping criterion. CH query algorithm adopts one-side-stopping criterion: when the distance of the settling node of one direction (i.e., one side) is larger than or equals to the best distance μ, then the query algorithm stops the search in this direction but continues searching in another direction until the direction also satisfies the stopping condition or the priority queue of the side is empty.

3.3.2 Pruning: search in ascending node order in two directions

CH query algorithm alternates search in the forward direction starting from the source node and in the reverse direction from the target node. In each direction, the current settling node only visits its nodes that have higher node orders and ignores whose orders are lower than it. The two direction could meet at somewhere (this case rarely happens in CH algorithm, though it almost always happens in the traditional bi-directional Dijkstra's algorithm), or one direction stops first and another continues until it satisfies the stopping condition or the priority queue is empty, shown as Figure 13.

A detailed proof the correctness of CH query algorithm can be found in [2]. Our report will not repeat the work, but we should point out that the proof needs a little improvement.

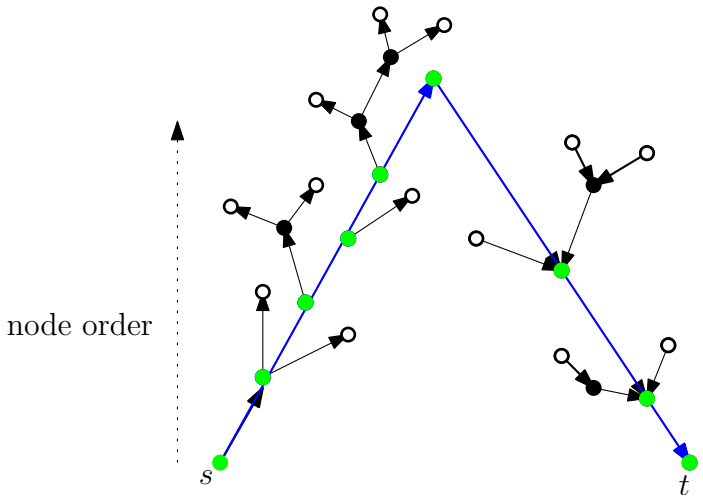

Figure 13: Search in ascending node order in two directions.

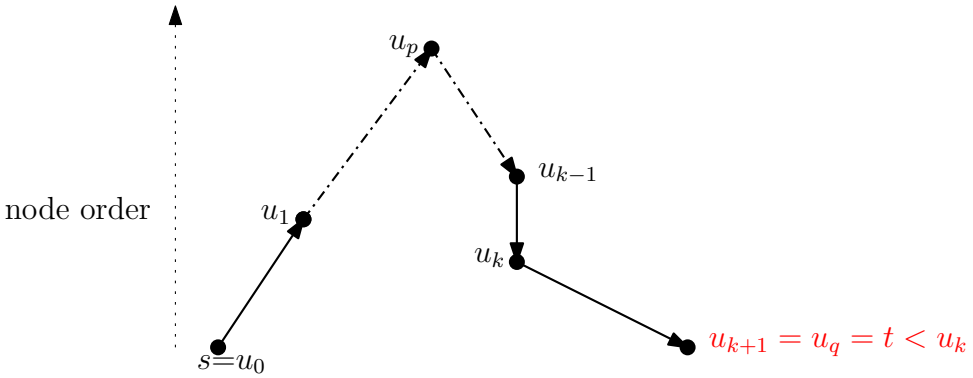

Figure 14: A special case in the proof of correctness of query in ascending node order [2].

In the proof, the special case in which $u_k + 1 = t$ is not considered carefully. It has two sub-cases. In sub-case 1 where $u_k < t$, it is just the case shown in that figure. But in sub-case 2 where $u_k + 1 = u_q = t < u_k$ as shown in Figure 14, we should not explain the correctness as these papers do, because t is contracted earlier than u_k and there are no shortcuts between $u_k - 1$ and t. Actually, in this sub-case, the CH query search from the reverse direction starting from t to u_k in ascending node order.

3.3.3 One-side-first stopping criterion of CH query algorithm

CH query algorithm applies the so called one-side-stopping criterion rather than the strong stropping criterion describered in Section 2.

Definition 17. *One-side stopping criterion:*

If $Q_f > \mu$, then stop the search in the forward direction;
If $Q_r > \mu$, then stop the search in the reverse direction.

Notice that the stopping criteria of the bi-directional Dijkstra's algorithm stop searches

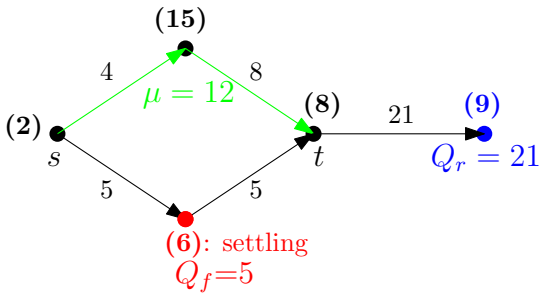

Figure 15: Counterexample: why CH query algorithm cannot use the strong stopping criterion.

in both sides (directions) simultaneously, but CH query algorithm stops side by side in different time. The proof of this stopping criterion is relatively easy.

Proof. (only in terms of the forward direction) Notice that the observations in the traditional bi-directional Dijkstra's algorithm still hold. Once $Q_f > \mu$, afterwards the distance from any settling node v to the source must be larger than the current μ because $d(s, v) = Q_f$ and Q_f is always increasing; moreover, the length of the path ℓ' from s to t via v is no less than $d(s, v)$. This means that $\ell' > \mu$ after the moment when $Q_f > \mu$, so μ will not be updated after that moment. □

The difficulty is the understanding why the strong stopping criterion used in the traditional bi-directional Dijkstra's algorithm becomes invalid when CH introduces the node order pruning technique. Here we restate the strong stopping criterion.

If $Q_f + Q_r \leq \mu$, then the algorithm can stop and return μ as the shortest distance $d(s, t)$.

In this report, we give a counterexample to illustrate why CH query algorithm cannot use the strong stopping criterion as shown in Figure 15 (the numbers in brackets denote nodes). The source node is 2 and the destination node is 8. The current best distance from node 2 to node 8 is $\mu = 4 + 8 = 12$ (the length of the path <2-15-8>). The currently $Q_f = 5$, $Q_r = 21$, so $Q_f + Q_r = 26 \geq \mu$, so the strong stopping criterion is satisfied; but we cannot stop now, because the final best distance $\mu = 5 + 5 = 10$ (the length of the path <2-6-8>).

However, the strong stopping criterion is not always gives wrong results. Our experiments show that the probability that it gives the correct distances is over 70% on the CH preprocessed file "USA-road-d.NY". Our experiments also show that the algorithm using the strong stopping criterion is about two times faster than that using the one-side stopping criterion.

3.4 Implementation of CH query algorithm

We have seen that the algorithm structure of CH query algorithm is quite similar to that of the bi-directional Dijkstra's algorithm. Basically it also uses double priority queues to store the nodes that have been visited but not yet settled. The settling node is the

CH query algorithm
Input: CH-preprocessed road networks file; source s and target t.
Output: the shortest distance $best = d(s,t)$ and the shortest path.

Node u, v
int *tentative* // tentative distance of (s,v)
int *dist* // the dist between the node and the respective source actually
int $best \leftarrow \infty$ // tentative optimal shortest distance of (s,t)
boolean $flag \leftarrow 1$ // turns to forward search(true) or reverse search(false)
int $Qr \leftarrow 0$, $QrDist \leftarrow 0$
BinaryHeap bheap[2]
bheap[flag].insert$(s, 0, flag)$ // insert source (s, 0) into binary heaps
node[s].dist[$flag$] $\leftarrow 0$ // tentative distance to s, or shortest distance to s after it is settled
node[s].bp[$flag$] $\leftarrow 1$ // the associated Binary Node ID and the distance of this node
node[s].state $\leftarrow$ SETTLED[$flag$]
bheap[$1 - flag$].insert$(t, 0, flag)$
node[t].dist[$1 - flag$] $\leftarrow 0$
node[t].bp[$1 - flag$] $\leftarrow 1$
node[t].state $\leftarrow$ SETTLED[$1 - flag$]
$dist \leftarrow 0$
while not bheap[$flag$].isEmpty() **or not** bheap[$1 - flag$].isEmpty())
    if bheap[$flag$].isEmpty()
        $flag \leftarrow 1 - flag$
    end if
    $u \leftarrow$ bheap[$flag$].removeMin($flag$)
    $dist \leftarrow$ node[u].dist[$flag$]
    node[u].state $\leftarrow$ SETTLED[$flag$]
    if $dist \geq best$ // One-side stopping criterion
        bheap[$flag$].clear()
        $flag \leftarrow 1 - flag$
        continue
    end if
    for each neighbor v of u:
        // Contraction Hierarchies pruning:
        // Search is in ascending order in both bi-directions.
        if node[v].order node[u].order
            next v
        end if
        if v.state = SETTLED[$flag$] // v has been settled in the flag direction
            next v
        end if
        $tentative \leftarrow dist$ + weight(u, v) // give tentative distance for v
(to be continued)

CH query algorithm (continued)

```
        if tentative v.dist[flag] // update v.dist[flag], and also update best
            if v is settled by 1 - flag direction
                if v.dist[flag] = ∞ // this implies v is UNREACHED on his direction
                    v.dist[flag] ← tentative
                    v.bp[flag] ← bheap[flag].insert(v, tentative, flag)
                    visited.push_back(v)
                else
                    v.dist[flag] ← tentative
                    bheap[flag].decreaseKey(v.bp[flag], tentative, flag)
                end if
                v.previous[flag] ← u // the immediate predecessor of v
                if v.dist[1 - flag ] ∞
                    if besttentative + v.dist[1 −flag]
                        best ← tentative + v.dist[1 −flag]
                    end if
                end if
            end if
        next v
    end for
    flag ← 1 − flag // turn to another searching direction
end while
showPath(s, t) // see algorithm showPath
return best
```

Table 1: typical number of average visited nodes and maximum heap sizes

	Avg visited nodes		max heap size	
	biDJ	CH	biDJ	CH
NY	83826	1496	2219	266
E	1312085	3579	6158	532

node which currently has the minimum distance to its starting node and it is popping out from the top of the priority queue. In a specific implementation of CH algorithm, different data structures of the priority queue result different performances. We use three data structures: a trivial vector; Fibonacci heaps [8] and binary heaps [9]. In the next sub-section, we will see the performance comparison of these three data structures on four different road networks under three different running environments.

The programming language we use is C++11.

3.4.1 a trivial vector

The trivial vector is very simple, but it is nearly as same efficient as Fibonacci heaps and binary heaps, so we need discuss it. It is defined as follows.

```
struct ODID {
    int ID // ID of node
    int OD // query order(prioirty) of node
}
vector < ODID > trivialHeap
```

Strictly speaking, $trivialHeap$ is not a priority queue, because the top of this vector is not the minimum node. But it supports all the operations including the three main operations of a priority queue: removeMin() insert() and decreaseKey(). We use only brute force search to retrieve specified nodes including the minimum node. So the time complexity of removeMin() and decreaseKey() are $O(|Q|)$, where $|Q|$ is the size of the heap (i.e., the vector), and the complexity of insert is $O(1)$ because we only put a visited node in the back of the vector if this node has not been visited before. The complexities of min() and isEmpty() are also $O(1)$.

Some typical number of average visited nodes and the maximum size of the priority queue by the bi-directional Dijkstra's algorithm and CH query algorithm on the file "USA-road-d.NY" and "USA-road-d.E" are as shown in Table 1.

Generally, the maximum heap size is a very small number compared to n, the number of all nodes on the road networks. Even if we test the bi-directional Dijkstra's algorithm on the file "USA-road-d.USA" where n is about 24 million, the maximum heap size is never over 10000 (in contrast, the visited nodes are up to the same order as n). This phenomenon explains why we can use the trivial vector as CH algorithm's data structure without losing much performance. This result is surprising. We can see this again in Section 4: Experiments.

3.4.2 Fibonacci heaps

1. Smaller and static root node array Fibonacci heaps is famous for its low time complexity on the three main operations required by Dijkstra's algorithm.

   The time complexity of its three main operations removeMin(), insert() and decreaseKey() are $O(\log |Q|)$, $O(1)$ and $O(1)$ [8], where $|Q|$ is the size of Fibonacci heaps. When it is applied to Dijkstra's algorithm, the time complexity of Dijkstra's algorithm reduces to $O(m+n\log n)$ [8]. If we carefully observe the time complexity, precisely it should be $O(m+n\log|Q|)$. We notice that the variable in logarithmic function is the heap size $|Q|$ rather than n; actually $|Q|$ is much smaller than n. We also notice that in the Fibonacci heaps design, the size of root node array $ArraySize$ is very small and 20 is large enough even for shortest path finding on "USA-road-d.USA", because roughly $2^{ArraySize}$ euqals the size of Fibonacci heaps, while the size of Fibonacci heaps is roughly $\sqrt{n}$ in the classical Dijkstra's algorithm and even smaller in CH query algorithm. Although $ArraySize$ is small, it impacts on the performance of the removeMin() operation (see Appendix: Code), so we should reduce its size as small as possible. To improve its performance, we define the root node array as a static array to avoid allocating memory frequently. These two measures slightly (about 10%) decrease the running time of CH query algorithm.

2. Fibonacci heaps is only efficient in theory, not in practice

   However, we notice that although theoretically Fibonacci heaps has the best time complexity among all the current data structures (see Table 4), practically it is of low efficiency because the removeMin() implementation involves as many as over 3000 C++ statements (we implement it in C++ and count the real number of statements in nested loops). The running time of so many statements is comparable to the brute force searching of the minimum node in the trivial vector of small size. This explains why complicated data structures such as Fibonacci heaps have only the nearly same performance as a trivial vector.

3.4.3 binary heaps

The third data structure we use to implement CH algorithm (as well as Dijkstra's algorithm) is binary heaps. The book [9] presents an excellent detailed description of binary heaps. As for binary heaps, the time complexity of removeMin(), insert() and decreaseKey() are all $O(\log |Q|)$. In CH query algorithm, because only a small number of nodes are visited, the size of the heap at any moment is actually a very small number, for example, several hundred. The time complexity of Dijkstra's algorithm is $O((m+n)\log n)$ [9, 14] (actually it is $O((m+n)\log|Q|)$; see CH query algorithm and the single-directional Dijkstra's algorithm). Thus compared to Fibonacci heaps, binary heaps are a little lower in performance in theory. But in practice, binary heaps outperform Fibonacci heaps because of the small size of heaps and the relatively simple implementation of binary heaps which involves much less C++ statements (less than tenth of statements the Fibonacci heaps implementation involves). Goldberg and Tarjan in 1996 [15] pointed out that "... the expected number of decrease-key operation in Dijkstra's shortest path algorithm is $O(n\log(1+m/n))$ ... this result explains why Dijkstra codes based on binary heaps per-

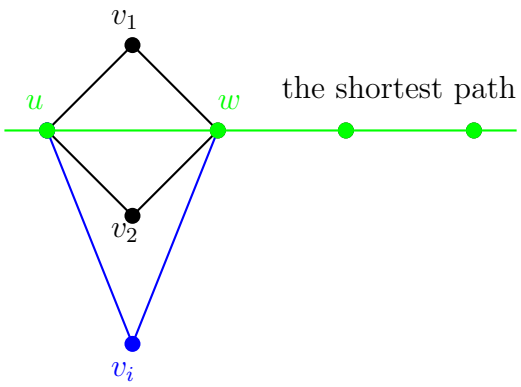

Figure 16: should $< u, v_i, w >$ be the in the unpacked path?

form better than ones on Fibonacci heaps." We independently discovered this fact.

3.4.4 Path unpacking

The path found by CH query algorithm comprises many shortcuts, recall that shortcuts are new added edges rather than original edges of the graph. However, what users need is the path comprising only the edges from the original graph (this path is named "the original path"). So we have to restore the path by removing shortcuts from the path and replacing them with the original edges of the graph. This process is path unpacking. It is like the reverse process of node contraction. The paper [1] presents a method to unpack a path, but it needs extra space which is less than the size of the original graph. We present a method to unpack a path without using extra space. Our method is based on the following observations (Figure 16).

1. For any two edges (u, v_i) and (v_i, w) of the graph with length ℓ = weight(u, v_i) + weight (v_i, w), if ℓ = weight(u, w), it is perhaps that the edge(u, w) is a shortcut and we can unpack it and obtain two edges (u, v_i) and (v_i, w); that is, we can insert v_i into the path and the new path is $< \ldots, u, v_i, w, \ldots >$.
2. If $\ell <$ weight(u, w) for any v_i, then we know that (u, w) is not a shortcut (because in such case we do not need to add shortcuts when we contracted v_i for all i); we do not need to add any node into the path for the edge (u, w).
3. It is impossible that $\ell <$ weight(u, w) (otherwise the edge (u, w) should not be on the path; it should be replaced by the two edges (u, v_i) and (v_i, w) or other some edges of which the weight sum is less than length). We can unpack the path iteratively as shown in the Figure 17 (contracted nodes $\{v, v', v''\}$ are added into the original path).

Considering the reverse process of node contraction, the algorithm of path showing and path unpacking are as follows (note that we need record the immediate predecessor node for each node in CH query algorithm. The predecessor is the current settling node).

The time complexity is $O(d^2|V'|)$, where d is the average degree of the graph and $|V'|$ the number of nodes that on the unpacked path. $|V'|$ is about $\sqrt{n_{visited}}$, where $n_{visited}$ is the number of nodes visited by CH algorithm. Experiments on the road networks "USA-

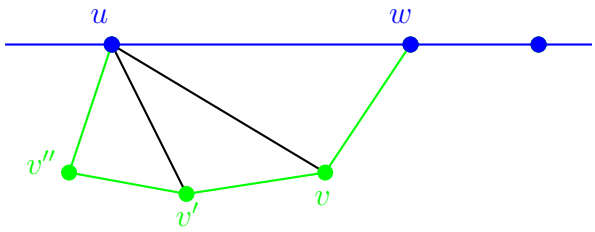

Figure 17: path unpacking iteratively

CH showPath algorithm

Input: the previous nodes of each node on path; the last node where two search trees meet and update the shortest distance μ

Output: nodes on the shortest path of the original graph

vector<**int**> onPath
int $temp \leftarrow lastMeetNode$ // $lastMeetNode$ is the last node updating μ before CH query stops
while $temp \neq s$, **do**
    onPath.push_back($temp$)
    $temp \leftarrow$ node[$temp$].previous[1]
end while
onPath.push_back(s)
reverse(onPath.begin(), onPath.end())
$temp \leftarrow lastMeetNode$
while $temp \neq t$, **do**
    $temp \leftarrow$ node[$temp$].previous[0]
    onPath.push_back($temp$)
end while
unpackPath(onPath)

CH path unpacking algorithm

Input: the shortest path represented in original edges + shortcuts; the last node $lastMeetNode$ where two search trees meet and update the shortest distance μ.
Output: the shortest path represented in only original edges.

```
Procedure unpackPath()
vector<int> originalPath
vector<int> contractedNodes // these nodes will be found and inserted into the path
int i, dist
Node v, u, w
adList p
for i = onPath.begin() to onPath.end() − 1, do
    v ← onPath[i]
    w ← onPath[i + 1]
    originalPath.push_back(v)
    contractedNodes.clear()
    while true, do
        dist ← weight(v, w)
        p ← v.adlist()
        u ← p.node
        while weight(v, u) + weight(u, w) ≠ dist, do
            next p
            u ← p.node
        end while
        if p ≠ NULL
            break
        end if
        contractedNodes.push_back(u)
        w ← u
    end while
    // insert the contracted nodes into original path
    while not contractedNodes.empty(), do
        originalPath.push_back(contractedNodes.back())
        contractedNodes.pop_back()
    end while
    next i
end for
originalPath.push_back(onPath.back()) // insert the last node on path
print originalPath[]
End Procedure
```

road-d.USA" show that path unpacking takes less than 2% of CH query time. This is ignorable compared to the time spent on path finding.

4 Experiments

4.1 Goal and design

As we have mentioned at the beginning of this report, our goal is to verify whether the performance of CH algorithm is as high as its inventors have claimed. Therefore our experiments are planned as follows:

1. Test the time consumption of CH algorithm on large road networks;
2. Compare it with the (single-directional and bi-directional) classical Dijkstra's algorithm.
3. Test the two algorithms on different machines with different operating system using different data structure, thus we can have various views to observe CH algorithm.
4. The source and the destination of a path are selected randomly. So the time used is an average value.

4.2 Data source and running environment

To have a full view of CH algorithm, we tested it under different OSs and hardware systems as follows.

1. Laptop + Windows 7

   Windows 7 Home Premium, Service Pack 1, 64-bit, only supports 31-bit (2GB) memory addressing ability; Visual C++ 6.0. Hardware: a laptop. Brand: Acer; Model: Aspire V5-571; CPU: Intel Core i3-3217U 1.80GHz; Memory: 4GB.

2. Laptop + Linux Ubuntu

   Ubuntu 13.04 and g++(GCC) 4.7.3. Ubuntu and Windows 7 run on the identical laptop (at different time).

3. Server CDMTCS + Linux Fedora

   Linux Fedora release 17 (Beety Miracle) with g++ (GCC) 4.7.2 on a local server named CDMTCS. CPU: Intel Core i7-2600 3.4GHz; Memory: 8GB, 8M cache, supporting 40-bit memory addressing ability.

The data sources we use are from [12]. The number of nodes and edges are in Table 2.

4.3 CH Preprocessing performance test

The papers [1, 2] present various combinations of priority terms for the node selecting order when we contract nodes at the preprocessing phase. We use a relatively simple combination: $E + D$, and our preprocessing time is very close to the time used by the same

Table 2: Data source files of road networks

Data source file	nodes	edges
USA-road-d.NY	264346	733844
USA-road-d.E	3598623	8778114
USA-road-d.W	6262104	15248146
USA-road-d.USA	23947347	58333344

Table 3: CH preprocessing experiment result

Data source file	Time	Memory	Shortcuts/Original edges	Query time	Avg visited nodes
USA-road-d.NY	147s	1.1GB	876894/733846	2.25ms	1803
USA-road-d.E	1406s	1.5GB	8510552/8778114	13.2ms	4268
USA-road-d.W	2467s	2.0GB	14685670/15248146	14.3ms	4619
USA-road-d.USA	12003s	4.8GB	58544880/58333344	84.45ms	13533

combination but it is longer than the time used by sophisticated combinations [1, 2]. It is still acceptable (3.3 hours for "USA-road-d.USA"). The comparison of a trivial algorithm to our preprocessing algorithm shows that the two strategies, restarting at stages and uniformity, are crucial for preprocessing performance as well as query performance. The following preprocessing experiments, which are based on the two strategies and the *Stage*[37] array as well as the single-source-multiple-destination Dijkstra's algorithm, are conducted on our CDMTCS server. Its results are shown in Table 3.

Note: "Time" and "Memory" are the preprocessing time and consumed memory respectively. The query time and th average visited nodes are tested on the preprocessed files. The CH query implementation is based on Binary heaps.

The trivial algorithm uses a simple restarting stage array which partitions all nodes evenly, that is, it restarts to recompute the order of the rest nodes after it has contracted the same number of nodes at each stage. It is stuck in the later preprocessing phase although it also uses the single-source-multiple-destination Dijkstra's algorithm to find witness paths and adopts the uniformity strategies. This means that the restarting strategy of CH preprocessing algorithm is very important.

4.4 CH Query performance test

The following table shows our experiment results of CH query efficiency test by comparing it with the traditional Dijkstra's algorithm and the bi-directional Dijkstra's algorithm on two hardware systems, three operating systems, and four data source files. These algorithm uses three different data structures: the trivial array, Fibonacci heaps and binary heaps. Our main results are shown in Table 4.

Note: "DJ" denotes the traditional single-directional Dijkstra's algorithm; "biDJ" denotes the bi-directional Dijkstra's algorithm; "spd" is the abbreviation of "speedup".

Table 4: Query performance comparison

		Laptop/Win7				Laptop/Ubuntu				CDMTCS/Fedora			
		DJ	biDJ	CH	spd	DJ	biDJ	CH	spd	DJ	biDJ	CH	spd
NY	B	120	80	4.7	17	160	100	4.7	21	100	50	2.3	22
	F	510	310	11.2	28	205	130	5.1	25	100	75	2.5	30
	T	240	210	7.4	28	535	455	5.4	84	200	300	2.8	107
E	B	1840	1540	43	36	2325	1775	27	66	1250	925	13	70
	F	23760	16700	121	138	2925	2275	28	80	1650	1225	15	82
	T	6380	5770	49	118	11800	20400	36	567	6100	11300	18	628
W	B	-	-	-	-	3990	3000	31	96	1875	1350	14	94
	F	-	-	-	-	5085	3990	33	121	2475	1800	17	107
	T	-	-	-	-	22925	29500	44	677	11500	16900	20	867
USA	B	-	-	-	-	-	-	-	-	8850	7525	85	**89**
	F	-	-	-	-	-	-	-	-	18025	9025	93	**98**
	T	-	-	-	-	-	-	-	-	116700	180900	143	1262

Each cell is the query time in millisecond; "NY", "E", "W" and "USA" are the source data files of which names remove "USA-road-d.". "B","F" and "T" refer to Binary heaps, Fibonacci heaps and the trivial array respectively.

According to the experiments, we can draw the following conclusions.

1. CH query algorithm is much faster than the bi-directional classical Dijkstra's algorithm. Generally, CH algorithm of our implementation is about two orders of magnification faster than the classical Dijkstra's algorithm. Our implementation has verified the efficiency of CH algorithm.
2. The speedup depends on the road network data sources, the data structures and even the running environment such as operating systems.
3. The famous data structures are not always more efficient than a trivial one. In CH algorithm, a trivial array with brute force searching on the array is competitive to binary heaps and Fibonacci heaps when it is applied to CH algorithm. This sounds amazing but it is supported by our experiments. This inspires us that the simplicity of implementation is also a very important aspect in algorithm and data structure design.
4. The bi-directional Dijkstra's algorithm is about $1.2-2$ times faster than the single directional version of this algorithm.

5 Conclusion and Future work

5.1 Our results

Our implementation of CH algorithm and the comparisons of CH query algorithm to the classical Dijkstra's algorithm by experiments have verified the efficiency of CH preprocessing algorithm and CH query algorithm.

The preprocessing time of CH algorithm, which is within a few hours on continental-sized road networks (see Table 3), is acceptable.

The preprocessing adds about the same number of shortcuts as the original edges.

The CH query algorithm, which can find a shortest path in 85ms on the whole USA road networks (see Table 4), is about two orders of magnitude faster than the classical Dijkstra's algorithm, better than 17 times speedup claimed by the paper [13] (although still lower than the speedup of four orders of magnitude claimed by the paper [1] and [2]).

We also observed that the speedup of CH to the bi-directional Dijkstra's algorithm depends on data structures and running environments (see Table 4). Sometimes these two factors have great influence on the speedup (but it has little impact on the performance of CH query algorithm).

Moreover, the time spent on path unpacking is negligible compared to the time spent on path finding and it needs no extra space overhead.

So from an overall view we have achieved our initial goal. Note that we had to design the implementation details because they are not presented in the literatures available.

5.2 Innovations

In our implementation of CH algorithm, we have some innovations summarized as follows.

1. We observe that Fibonacci heaps has only low time complexity in theory but practically it is not advantageous in efficiency because its implementation involves too many machine instruments. We also observed that the size of Fibonacci heaps is very small when CH query algorithm is running. So we inferred that a trivial array can reach the similar performance of Fibonacci heaps. Our experiments have verified this surprising idea. We noticed that our idea is consistent with [15].

   We also point out that, suppose we use binary heaps as the underlying data structure, the time complexity of Dijkstra's algorithm, although claimed to be $O((m+n)\log n)$ in many literatures(e.g., [9] and [14]), precisely it should be $O((m+n)\log|Q|)$, where $|Q|$ is the size of the priority queue and it is much smaller than n.

2. We propose an exponentially decreasing stage array $Stage[37]$ as restarting points of node ordering selection for preprocessing phase. With this stage array, we realized CH preprocessing algorithm using relatively simple priority term combination.

3. To improve the performance of witness path finding when we contract nodes at the CH preprocessing phase, we proposed a method using the single-source-multiple-destination Dijkstra's algorithm to compute shortest distances simultaneously for

multiple node pairs. It is d (d is the average degree of the network graph) times faster than so called the bucket method using limited hops (1 hop, 2 hops,..., up to 6 hops) of local search [1]. The advantage of our method lies in that it avoids much repeated computation.

4. This report strengthens the theory foundations of the bi-directional Dijkstra's algorithm and CH query algorithm. We present a detailed and straightforward proof for the stopping criterion of the bi-directional Dijkstra's algorithm; we point out that so called the weak stopping criterion is not strictly correct. We also remedy a missing point in the original correctness proof of CH query algorithm from its inventors. Moreover, we explain why the stopping criterion of the bi-directional Dijkstra's algorithm cannot be directly used in CH query algorithm.

5. By the observation that a shortcut is added to replace two edges which have the same length sum as the shortcut, we present an efficient solution of path unpacking that does not have the need for storing extra information. In contrast, the inventors of CH algorithm use extra space overhead for path unpacking [1, 2].

6. We found another novel method to search the shortest distances during this project (see next subsection: Future work).

5.3 Future work

However, we still have some work worthy of doing in the future. At first, our preprocessing uses relatively simple priority terms for node selection ordering during the node contraction process, thus the preprocessing time and query time of CH algorithm are more than that of the algorithm's inventors. It is possible to achieve better results but it requires careful tune of coefficients of priority terms and proper combinations of these priority terms. This can be done together with a hardware-accelerating approach presented by Delling et al. [13] which uses the parallelism of multi-core CPU and GPUs to speedup CH query algorithm. However, we do not have enough time to finish this work in few weeks.

Secondly, we observed that although the strong stopping criterion has one forth probability of failure when it is adopted by CH query algorithm, it can speedup CH query algorithm two times, so it is worthy of studying how to modify CH algorithm so that it can be compatible with the strong stopping criterion.

Finally, during the work of this project, we found a novel way to search shortest paths. This way also involves two phases: the preprocessing phase and the query phase. The preprocessing phase partitions the graph into many small blocks and assigns each node coordinates (x, y, z, w). The query phase is based on the single-directional (or bi-directional) Dijkstra's algorithm, but the new algorithm only visits nodes whose coordinates are between the coordinates of the source and the destination. Specifically, if the source's coordinate is (x_1, y_1, z_1, w_1), and the destination's coordinate is (x_2, y_2, z_2, w_2), then it only visits the nodes of which coordinates are within ($[x_1, x_2]$, $[y_1, y_2]$, $[z_1, z_2]$, $[w_1, w_2]$). Those nodes out of this range are pruned. Obviously, this algorithm reduces much search space. Maybe we can even combine the new algorithm with CH algorithm to gain more performance improvement.

6 Acknowledgement

First of all, the author thanks my supervisor, Dr. Michael J. Dinneen. He has given me many advices and suggestions, and more importantly, without his encouragement I would have given up this project at the initial phase. Also I thank him for his CD, candy, wine and his story about him and Edward Teller at the Los Alamos National Laboratory. Anyway, it was Michael who led me on this busy, interesting and exciting research travel.

The author also thanks my friends Lucas Yang, Ricky Shu and Kuai Wei for their beneficial discussions and kind help.

References

[1] Geisberger, R. (2008). Contraction hierarchies: Faster and simpler hierarchical routing in road networks. Diploma Thesis.

[2] Geisberger, R., Sanders, P., Schultes, D. and Vetter, C. (2012). Exact routing in large road networks using contraction hierarchies. Transportation Science, vol.46(3), pp. 388-404.

[3] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische mathematik, vol.1(1), pp. 269-271.

[4] Pohl, I. (1969). Bi-directional and heuristic search in path problems (Doctoral dissertation, Department of Computer Science, Stanford University)

[5] Geisberger, R., Sanders, P., Schultes, D. and Delling, D. (2008). Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In Experimental Algorithms (pp. 319-333). Springer Berlin Heidelberg.

[6] Schultes, D. (2008, February). Route Planning in Road Networks. In Ausgezeichnete Informatikdissertationen.

[7] Abraham, I., Fiat, A., Goldberg, A. V. and Werneck, R. F. (2010, January). Highway dimension, shortest paths, and provably efficient algorithms. In Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (pp. 782-793). Society for Industrial and Applied Mathematics.

[8] Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM (JACM), vol. 34(3), pp. 596-615.

[9] Dinneen, M.J., Gimel'farb G., and Wilson, M.C. (2013). An Introduction to Algorithms, Data Structures and Formal Languages. (3nd edition, ebook), pp. 153-157.

[10] Geisberger, R. and Schieferdecker, D. (2010, August). Heuristic Contraction Hierarchies with Approximation Guarantee. In Third Annual Symposium on Combinatorial Search.

[11] Goldberg, A., Kaplan, H. and Werneck, R. (2005). Point-to-point shortest path algorithm. U.S. Patent Application 11/321, p. 349.

[12] U.S. Census Bureau:
http://www.census.gov/geo/maps-data/data/tiger-geodatabases.html

[13] Delling, D., Goldberg, A. V., Nowatzyk, A. and Werneck, R. F. (2012). PHAST: Hardware-accelerated shortest path trees. Journal of Parallel and Distributed Computing.

[14] Barbehenn, M. (1998). A note on the complexity of Dijkstra's algorithm for graphs with weighted vertices. Computers, IEEE Transactions on, vol. 47(2), p.263.

[15] Goldberg, A. V. and Tarjan, R. E. (1996). Expected performance of Dijkstra's shortest path algorithm. NEC Research Institute Report.

[16] Goldberg, A. V., Kaplan, H. and Werneck, R. F. (2006, January). Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In Alenex (Vol. 6, No. 2, pp. 129-143).

[17] Goldberg, A. V. and Werneck, R. F. F. (2005, January). Computing Point-to-Point Shortest Paths from External Memory. In Alenex/Analco (pp. 26-40).

Appendix: Code

This code has been successfully compiled under Windows 7 Home Premium (Service Pack 1) plus Visual C++ 6.0, Linux Ubuntu 13.04 (compiling command: g++ -std=c++11 o ch CH.cpp) with g++ (GCC) 4.7.3, and Linux Fedora release 17 (Beety Miracle) with g++ (GCC) 4.7.2.

(Our C++11 code is in the source file CH.cpp.)